StoreGate: a Data Model for the ATLAS Software Architecture

P. Calafiura[1], H. Ma[2], M. Marino[1], S. Rajagopalan[2] and D. R. Quarrie[1]
[1](Lawrence Berkeley National Lab, U.S.A.)
[2](Brookhaven National Lab, U.S.A.)

Abstract

ATLAS[1] has recently joined Gaudi, an open project to develop a data processing framework for HEP experiments[2]. The data model is one of the areas where ATLAS has extended more the original Gaudi design to meet the experiment's own requirements. This paper describes StoreGate, the first implementation of the ATLAS Data Model.

Keywords: transient data model, template meta-programming

1 Introduction

The Gaudi software architecture belongs to the blackboard family[3]: data objects produced by knowledge modules (called Algorithms in Gaudi) are posted to a common "in-memory data base" from where other modules can access them and produce new data objects.

This model greatly reduces the coupling between knowledge modules containing the algorithmic code for analysis and reconstruction, in that one knowledge module does not need anymore to know which specific module can produce the information it needs nor which protocol it must use to obtain it (the "interface explosion" problem described in component software systems). Algorithmic code is known to be the least stable component of software systems and the blackboard approach has been very effective at reducing the impact of this instability, from the Zebra system of the FORTRAN days to the Java InfoBus architecture.

The Transient Data Store (TDS) is the blackboard of the Gaudi architecture: a module creates a data object and post it to the TDS to allow other modules to access it[1].

Once an object is posted on to the store, the TDS takes ownership of it and manages its lifetime according to preset policies, removing, for example, a TrackCollection when a new event is read. The TDS also manages the conversion of a data object from/to its persistent form and provides therefore an API to access data stored on persistent media.

2 StoreGate Design and Functionality

StoreGate (SG), in common with most other existing data models, is basically a dictionary of data objects which manages their memory and oversees conversion to/from persistency. The SG design process has been informal and iterative. We released early and often and used developers feedback to adjust our initial design concept[2]. The result may lack the coherency of a formal top-down design but it follows a few principles which have proved to be useful.

Work with User Types

The success of the STL and of other public domain template libraries means that it has become vital to design an open system that can work with generic types that export an interface, in particular the STL containers, rather than forcing data objects to import a common interface.

[1]to be precise the current TDS implements only a "passive" blackboard, since modules do not react to TDS events (e.g. executing after a data object is registered into the TDS)

[2]which was in any case largely based on ideas which have worked in existing data models

SG adapts its behavior to the functionality each data object exports. The only SG-imposed constraint on a data object[3] is to be an STL *Assignable* type[4].

Avoid User-defined Keys

The disadvantage of the data/knowledge objects separation is the need for knowledge objects to identify data objects to be posted on or retrieved from the blackboard. It is crucial to develop a data model optimized for the required access patterns and yet flexible enough to accommodate the unexpected ones.

SG addresses this problem with a two-step approach: it defines a natural identifier mechanism for data objects and it transparently associates to each data object a default value of this identifier allowing developers to register and retrieve data objects without having to identify them explicitly.

The first component of the identifier is the data object type. Experience shows that HEP developers tend to group the objects they work on into collections. As a result the TDS will often contain a single instance of a data object type (say a `TrackCollection` or several closely related ones (e.g. a TrackCollection for each component of the Inner Detector). The SG retrieve interface covers these two use cases

```
DataHandle<TrackCollection> theTrackColl; //STL forward_iterator
sg->retrieve(theTrackColl);  //get the (default) TrackCollection
DataHandle<TrackCollection> beginTrackColls, endTrackColls;
sg->retrieve(beginTrackColls, endTrackColls); //get all TrackColls
```

Type-based identification is not always sufficient. For example the TDS may contain several equivalent instances of a TrackCollection produced by alternative tracking algorithms. Therefore we need to add a second component to our identification mechanism: the identifier of the Algorithm instance that produced the data object we want[4]. In the spirit of working with user types, the SG will allow developers to augment this history identifier with a generic key type optimized for their access patterns.

Control Object Access and Creation

The TDS is the main channel of communication among modules. A data object is often the result of a collaboration among several modules. SG allows a module to use transparently a data object created by an upstream module or read from disk.

A *Virtual Proxy*[5] defines and hides the cache-fault mechanism: upon request[5], a missing data object instance can be transparently created and added to the TDS, presumably retrieving it from a persistent data-base or, in principle, even reconstructing it on demand.

To ensure reproducibility of data processing, a data object should not be modified after it has been published to the store, we use the same proxy scheme to enforce an "almost const" access policy: modules downstream of the publisher are only allowed to retrieve a constant iterator to the published object.

Support Inter-object Relationships

SG supports uni-directional inter-objects relationships, or links, and will support bi-directional links in the future. A link is a persistable pointer. If the linked object is a data object then the proxy scheme described above is also used to implement the link. But typically links will refer

[3]this does not mean that the data model, simulation and reconstruction groups should not issue design guidelines to ensure that ATLAS data objects behave consistently in terms of memory management and persistability

[4]notice that we need to identify the instance rather than the class. In an often quoted use case, clients may want to distinguish among tracks reconstructed by the same tracking algorithm using different jet cone sizes.

[5]Currently the proxy uses lazy instantiation (i.e. the object is created only when the handle is dereferenced).

to objects that are not data objects but are contained within a data object. The SG knows how to get to the container and the container knows how to return an element given its index. The job of the link is to find out the value of the index, persistify it and, later on, pass it on to the container and get back the linked object. In the next section we will discuss how links handle indices into generic containers.

3 Implementation Techniques

A big advantage that SG has compared to earlier data models implementations is that many compilers are catching up with the ISO/ANSI C++ standard. Because of that, a new generation of template libraries like boost[6] and loki[7] are bringing once-esoteric techniques like template meta-programming into the mainstream. Template meta-programming uses the compiler template expansion to control and generate running code based on static type information. In SG we have used some of its simpler techniques.

Type Traits and Traits Types

The TDS memory management back-end manages the data objects as instances of a `DataObject` base class. Each class derived from `DataObject` has a unique `ClassID`. This allows, for example, to use an *Abstract Factory*[5] to create data object instances when reading from disk. SG wraps each stored data object into a templated `DataObject`

```
 template <typename DOBJ> class DataBucket : public DataObject {...}
```

If `DOBJ` does not inherit from `DataObject` we want the developer to define a `ClassID` for `DOBJ` that we will associate to the data object.

To determine, at compile time, if `DOBJ` inherits from `DataObject` we use the boost type trait `boost::is_convertible<DOBJ,DataObject>`, a template that evaluates to true when `DOBJ` can be assigned as a `DataObject`[6, 7].

To associate the `ClassID` information to a data object type, say vector<`double`>, we define a `ClassID_traits` structure that developers specialize for that data object (the struct is actually generated using a cpp macro)

```
 template <> struct ClassID_traits<vector<double> > {
    typedef type_tools::true_tag has_classID_tag;
    static const int ID = 1234;
    ....
 };
```

Concept Checking

SG allows developers to use generic key types to identify objects of a given type. A key must of course define an ordering operation. For SG we also require keys to be persistable. In traditional OO programming these requirements would be expressed as an interface the key class imports. In generic programming interfaces are rather exported and hence verified by the clients. To this end, SG provides a `KeyConcept` built using the boost `concept_check` library

```
  template <typename T, .... > struct KeyConcept {
    void constraints() {
      boost::function_requires< boost::LessThanComparableConcept<T> >();
      ....
    }
  };
```

Inserting a call to `boost::function_requires<KeyConcept<KEY> > ()` we allow the compiler to check whether the template parameter `KEY` of a retrieve or register method is valid.

Policy Classes

SG handle and link classes use policy classes to configure their behavior at compile time. A policy[7] is a statically configured *Strategy*[5]. It can also be seen as a traits class that defines behavior rather than structure. Policies become powerful tools when they are combined: the compiler picks the right combinations and generates the code needed by the application. For example the link class template `DataLink` is implemented as a combination of two policies

```
template <typename DOBJ,
          class IndexingPolicy=NoIndexing<DOBJ>,
          template <class> class DataStoragePolicy=DataHandleStorage>
class DataLink : public DataStoragePolicy<DOBJ>,
                 public IndexingPolicy  { ... }
```

`DataStoragePolicy` wraps the TDS back-end API, while `IndexingPolicy` defines the strategy the `DataLink` uses to find a container element given its identifier, and viceversa. We have defined indexing policy classes that can be used to index elements of all STL containers and to index nodes of an HepMC graph[8]. Policies are flexible: if a developer introduces a new container type, all they have to do is to provide a matching indexing policy and the compiler will generate the new link type as needed.

4 Status and Outlook

After more than one year of evolution the StoreGate design has achieved a certain maturity. A lot of broad design principles have been established: work with user types, avoid user-defined keys, define an access control policy. The core data access API has been stable for several months. Many packages have ported their code to StoreGate or are in the process of doing so. We are now working on aspects of the implementation and performance that are not yet production quality.

In the spirit of the Gaudi open project we have started discussing our work with the Gaudi community and we hope the StoreGate ideas and code will be useful to developers inside and outside ATLAS.

Acknowledgments

We would like to thank all ATLAS collaborators who contributed to the design and prototyping of SG. We are extremely grateful to many colleagues from other experiments who shared their experiences with us: M. Frank, V. Innocente, R. Kennedy, J. Kowalkowski, P. Mato, M. Paterno, S. Patton, S. Snyder and L. Tuura.

References

[1] W.W. Armstrong *et al.* , “ATLAS Technical Proposal”, CERN/LHCC/94-43.

[2] M. Cattaneo *et al.* , “Status of the GAUDI event-processing framework”, this conference. `http://proj-gaudi.web.cern.ch/proj-gaudi`

[3] F. Buschmann *et al.*, “Pattern-Oriented Software Architecture”, Wiley, 1996

[4] `http://www.sgi.com/tech/stl`

[5] E. Gamma *et al.*, “Design Patterns”, Addison-Wesley, 1994.

[6] `http://www.boost.org`

[7] A. Alexandrescu, “Modern C++ Design”, Addison-Wesley, 2001 `http://www.moderncppdesign.com`

[8] `http://mdobbs.home.cern.ch/mdobbs/HepMC/`